## **vn.py快速入门6 - 开发第一个量化策略**

原创 用Python的交易员 VNPY  2019-08-01

收录于话题

#vn.py快速入门

8个

快速入门系列已经到了第6篇，终于要开始接触编程实操的内容了。这篇教程中的内容，假设你对于Python语言的开发已经有了一定的基础掌握：

了解Python的数据结果

理解面向对象编程的概念

能使用控制语句来表述逻辑



如果没有也没关系，推荐一本快速入门的书《笨方法学Python3》（Learn Python the Hard Way），也是我自己多年前从完全0基础的小白第一次学习Python的敲门砖，50多堂课从头到尾敲一遍，只要不偷懒包敲包会，不会来找我。

**准备IDE工具**

用电脑写文章通常需要Word，同样写程序代码也需要专用的工具，这种工具就叫做IDE，全称Integrated Development Environment，中文名叫做“集成开发环境”。

IDE提供了一整套包括代码编写、调试运行、版本控制、管理界面等等写程序时所必须的工具环境，接下来教程里我们选择使用的是Visual Studio Code（简称VS Code），由微软推出的编程专用开源编辑器（然后被热心的开源社区加上各种插件打造成了超级IDE）。

前往VSCode首页：<https://code.visualstudio.com/>，点击Download绿色图标下载后，一路傻瓜安装，运行后看到下述界面：



在左侧导航栏顶部的5个按钮中，点击最下方的按钮进入安装扩展插件的页面，在搜索框中输入Python回车，找到由微软官方推出的Python插件：



点击上图红框中的绿色安装按钮，会自动在后台完成插件的下载安装启动，至此就已经准备好了我们的IDE。

尽管这里我们推荐使用VS Code，但如果你已经有了常用的IDE工具，不管是PyCharm、WingIDE、Vim还是Visual Studio，都可以用来完成后续的策略代码开发工作，所以直接用就好。假设过程中遇到了某些难以解决的问题，可以再换回到VS Code，本系列教程中的操作保证都完全可用。

**创建策略文件**

首先要接触的是一个用户目录的概念，即任何操作系统默认用来存放当前登录的用户运行程序时缓存文件的目录，假设你的登录用户名为client，那么：

Windows下：C:\Users\client\

Linux或Mac下：/home/client/

上述即为最常用的用户目录路径，注意只是常用情况，如果你的系统做了特殊配置修改可能不同。

VN Trader默认的运行时目录即为操作系统用户目录，启动后会在用户目录下创建.vntrader文件夹用于保存配置和临时文件（有时遇到奇怪的bug，删除该文件夹后重启就能解决）。

同时CTA策略模块（CtaStrategyApp）在启动后，也会扫描位于VN Trader运行时目录下的strategies文件夹来加载用户自定义的策略文件，以Windows为例：C:\Users\client\strategies。注意strategies文件夹默认是不存在的，需要用户自行创建。

进入strategies目录后，新建我们的第一个策略文件：demo\_strategy.py，然后用VS Code打开。

**定义策略类**

新建的策略文件打开后，内部空空如也，此时我们开始往其中添加代码，遵循行业传统，这里也同样选择用傻瓜的双均线策略作为演示：

from vnpy.app.cta\_strategy import ( CtaTemplate, StopOrder, TickData, BarData, TradeData, OrderData, BarGenerator, ArrayManager,)  
  
class DemoStrategy(CtaTemplate): """演示用的简单双均线"""  
 *# 策略作者* author = "Smart Trader"  
 *# 定义参数* fast\_window = 10 slow\_window = 20  
 *# 定义变量* fast\_ma0 = 0.0 fast\_ma1 = 0.0 slow\_ma0 = 0.0 slow\_ma1 = 0.0  
 *# 添加参数和变量名到对应的列表* parameters = ["fast\_window", "slow\_window"] variables = ["fast\_ma0", "fast\_ma1", "slow\_ma0", "slow\_ma1"]  
 def \_\_init\_\_(self, cta\_engine, strategy\_name, vt\_symbol, setting): """""" super().\_\_init\_\_(cta\_engine, strategy\_name, vt\_symbol, setting)  
 *# K线合成器：从Tick合成分钟K线用* self.bg = BarGenerator(self.on\_bar)  
 *# 时间序列容器：计算技术指标用* self.am = ArrayManager()  
 def on\_init(self): """ 当策略被初始化时调用该函数。 """ *# 输出个日志信息，下同* self.write\_log("策略初始化")  
 *# 加载10天的历史数据用于初始化回放* self.load\_bar(10)  
 def on\_start(self): """ 当策略被启动时调用该函数。 """ self.write\_log("策略启动")  
 *# 通知图形界面更新（策略最新状态）* *# 不调用该函数则界面不会变化* self.put\_event()  
 def on\_stop(self): """ 当策略被停止时调用该函数。 """ self.write\_log("策略停止")  
 self.put\_event()  
 def on\_tick(self, tick: TickData): """ 通过该函数收到Tick推送。 """ self.bg.update\_tick(tick)  
 def on\_bar(self, bar: BarData): """ 通过该函数收到新的1分钟K线推送。 """ am = self.am  
 *# 更新K线到时间序列容器中* am.update\_bar(bar)  
 *# 若缓存的K线数量尚不够计算技术指标，则直接返回* if not am.inited: return  
 *# 计算快速均线* fast\_ma = am.sma(self.fast\_window, array=True) self.fast\_ma0 = fast\_ma[-1] *# T时刻数值* self.fast\_ma1 = fast\_ma[-2] *# T-1时刻数值*  
 *# 计算慢速均线* slow\_ma = am.sma(self.slow\_window, array=True) self.slow\_ma0 = slow\_ma[-1] self.slow\_ma1 = slow\_ma[-2]  
 *# 判断是否金叉* cross\_over = (self.fast\_ma0 > self.slow\_ma0 and self.fast\_ma1 < self.slow\_ma1)  
 *# 判断是否死叉* cross\_below = (self.fast\_ma0 < self.slow\_ma0 and self.fast\_ma1 > self.slow\_ma1)  
 *# 如果发生了金叉* if cross\_over: *# 为了保证成交，在K线收盘价上加5发出限价单* price = bar.close\_price + 5  
 *# 当前无仓位，则直接开多* if self.pos == 0: self.buy(price, 1) *# 当前持有空头仓位，则先平空，再开多* elif self.pos < 0: self.cover(price, 1) self.buy(price, 1)  
 *# 如果发生了死叉* elif cross\_below: price = bar.close\_price - 5  
 *# 当前无仓位，则直接开空* if self.pos == 0: self.short(price, 1) *# 当前持有空头仓位，则先平多，再开空* elif self.pos > 0: self.sell(price, 1) self.short(price, 1)  
 self.put\_event()  
 def on\_order(self, order: OrderData): """ 通过该函数收到委托状态更新推送。 """ pass  
 def on\_trade(self, trade: TradeData): """ 通过该函数收到成交推送。 """ *# 成交后策略逻辑仓位发生变化，需要通知界面更新。* self.put\_event()  
 def on\_stop\_order(self, stop\_order: StopOrder): """ 通过该函数收到本地停止单推送。 """        pass

在文件头部我们的一系列import中，最重要的就是CtaTemplate，这是我们开发CTA策略所用的**策略模板基类**，策略模板提供了一系列以**on\_**开头的回调函数，用于接受事件推送，以及其他主动函数用于执行操作（委托、撤单、记录日志等）。

所有开发的策略类，都必须继承CtaTemplate基类，然后在需要的回调函数中实现策略逻辑，即当某件事情发生时我们需要执行的对应操作：比如当收到1分钟K线推送时，我们需要计算均线指标，然后判断是否要执行交易。

**设置参数变量**

所有的量化交易策略必然都会涉及到参数和变量这两个和数值相关的概念。

**参数**是策略内部的逻辑算法中用来控制结果输出的一些数值，在策略类中需要定义出这些参数的默认数值：

*# 定义参数*fast\_window = 10slow\_window = 20

定义完后，还需要参数的名称（字符串）添加到parameters列表中：

parameters = ["fast\_window", "slow\_window"]

这一步操作是为了让系统内的策略引擎，得以知道该策略有哪些参数，并在初始化策略时弹出相应的对话框让用户填写，或者在命令行模式下直接将配置字典中对应的key的value赋值到策略变量上。

**变量**则是策略内部的逻辑算法在执行过程中用来缓存中间状态的一些数值，在策略类中同样需要定义出这些变量的默认数值：

*# 定义变量*fast\_ma0 = 0.0fast\_ma1 = 0.0slow\_ma0 = 0.0slow\_ma1 = 0.0

定义完后，还需要变量的名称（字符串）添加到variables列表中：

variables = ["fast\_ma0", "fast\_ma1", "slow\_ma0", "slow\_ma1"]

和参数类似，这一步操作是为了让系统内的策略引擎，得以知道该策略有哪些变量，并在GUI图形界面上更新策略状态时将这些变量的最新数值显示出来，同时在保存策略运行状态到缓存文件中时将这些变量写入进去（实盘中每天关闭策略时会自动缓存）。

需要注意的是：

无论变量还是参数，都**必须定义在策略类中**，而非策略类的\_\_init\_\_函数中；

参数和变量，均只支持Python中的四种基础数据类型：**str、int、float、bool**，使用其他类型会导致各种出错（尤其注意不要用list、dict等容器）；

如果在策略逻辑中，确实需要使用list、dict之类的容器用于数据缓存，**请在\_\_init\_\_函数中创建这些容器。**

**交易逻辑实现**

上面已经提到，在vn.py的CTA策略模块中，所有的策略逻辑都是由事件来驱动的。对于事件，举例来说：

策略被执行初始化操作，会收到on\_init函数的调用，此时可以加载历史数据，来进行技术指标的初始化运算

新一根1分钟K线走完时，会收到on\_bar函数的调用，参数为该根K线对象的BarData

策略发出的委托，状态发生变化时，会收到on\_order函数的调用，参数为该委托的最新状态OrderData

对于最简单的双均线策略的DemoStrategy来说，我们不用关注委托状态变化和成交推送之类的细节，只需要在收到K线推送时（on\_bar函数中）执行交易相关的逻辑判断即可。

每次新的一根K线走完时，策略会通过on\_bar函数收到该根K线的数据推送。注意此时收到的数据只有该K线，但大部分技术指标计算时都需要过去N个周期的历史数据。

所以为了计算均线技术指标，我们需要使用一个叫做时间序列容器ArrayManager的对象，用于实现了K线历史的缓存和技术指标计算，该对象的创建，在策略的\_\_init\_\_函数中：

# 时间序列容器：计算技术指标用self.am = ArrayManager()

在on\_bar函数的逻辑中，第一步需要将K线对象推送到该时间序列容器中：

*# 纯粹为了后续可以少写一些self.*am = self.am  
*# 更新K线到时间序列容器中*am.update\_bar(bar)  
*# 若缓存的K线数量尚不够计算技术指标，则直接返回*if not am.inited: return

为了满足技术指标计算的需求，我们通常需要最少N根K线的缓存（N默认为100），在推送进ArrayManager对象的数据不足N之前，是无法计算出需要的技术指标的，对于缓存数据是否已经足够的判断，通过am.inited变量可以很方便的判断，在inited变为True之前，都应该只是缓存数据而不进行任何其他操作。

当缓存的数据量满足需求后，我们可以很方便的通过am.sma函数来计算均线指标的数值：

*# 计算快速均线*fast\_ma = am.sma(self.fast\_window, array=True)self.fast\_ma0 = fast\_ma[-1] *# T时刻数值*self.fast\_ma1 = fast\_ma[-2] *# T-1时刻数值*  
*# 计算慢速均线*slow\_ma = am.sma(self.slow\_window, array=True)self.slow\_ma0 = slow\_ma[-1]self.slow\_ma1 = slow\_ma[-2]

注意这里我们传入了可选参数array=True，因此返回的fast\_ma为最新移动平均线的数组，其中最新一个周期（T时刻）的移动均线ma数值可以通过-1下标获取，上一个周期（T-1时刻）的ma数值可以通过-2下标获取。

有了快慢两根均线在T时刻和T-1时刻的数值后，我们就可以进行双均线策略的核心逻辑判断，即是否发生了均线金叉或者死叉：

*# 判断是否金叉*cross\_over = (self.fast\_ma0 > self.slow\_ma0 and self.fast\_ma1 < self.slow\_ma1)  
*# 判断是否死叉*cross\_below = (self.fast\_ma0 < self.slow\_ma0 and self.fast\_ma1 > self.slow\_ma1)

所谓的均线金叉，是指T-1时刻的快速均线fast\_ma1低于慢速均线slow\_ma1，而T时刻时快速均线fast\_ma0大于或等于慢速均线slow\_ma10，实现了上穿的行为（即金叉）。均线死叉则是相反的情形。

当金叉或者死叉发生后，则需要执行相应的交易操作：

# 如果发生了金叉if cross\_over: # 为了保证成交，在K线收盘价上加5发出限价单 price = bar.close\_price + 5  
 # 当前无仓位，则直接开多 if self.pos == 0: self.buy(price, 1) # 当前持有空头仓位，则先平空，再开多 elif self.pos < 0: self.cover(price, 1) self.buy(price, 1)  
# 如果发生了死叉elif cross\_below: price = bar.close\_price - 5  
 # 当前无仓位，则直接开空 if self.pos == 0: self.short(price, 1) # 当前持有空头仓位，则先平多，再开空 elif self.pos > 0: self.sell(price, 1) self.short(price, 1)

对于简单双均线策略来说，用于处于持仓的状态中，金叉后拿多仓，死叉后拿空仓。

所以当金叉发生时，我们需要检查当前持仓的情况。如果没有持仓（self.pos == 0），说明此时策略刚开始交易，则应该直接执行多头开仓操作（buy）。如果此时已经持有空头仓位（self.pos , 0），则应该先执行空头平仓操作（cover）然后同时立即执行多头开仓操作（buy）。为了保证成交（简化策略），我们在下单时选择了加价的方式来实现（多头+5，空头-5）。

注意尽管这里我们选择使用双均线策略来做演示，但在实践经验中简单均线类的策略效果往往非常差，千万不要拿来跑实盘，也不建议在此基础上进行扩展开发，毕竟在豆腐渣工程上建什么都还是豆腐渣......

**实盘K线合成**

DemoStrategy的双均线交易逻辑，可以通过超价买卖的方式来保证成交，从而忽略撤单、委托更新、成交推送之类更加细节的事件驱动逻辑。

但在实盘交易时，任何交易系统（不管是期货CTP、还是数字货币BITMEX等）都只会推送最新的Tick更新数据，而不会有完整的K线推送，因此用户需要自行在本地完成Tick到K线的合成逻辑。

这块vn.py也提供了完善的K线合成工具BarGenerator，用户只需在策略的\_\_init\_\_函数中创建实例：

# K线合成器：从Tick合成分钟K线用self.bg = BarGenerator(self.on\_bar)

其中BarGenerator对象创建时，传入的参数（self.on\_bar）是当1分钟K线走完时触发的回调函数。

在实盘策略收到最新的Tick推送时，我们只需要将TickData更新到BarGenerator中：

def on\_tick(self, tick: TickData): """ 通过该函数收到Tick推送。 """ self.bg.update\_tick(tick)

当BarGenerator发现某根K线走完时，会将过去1分钟内的Tick数据合成的1分钟K线推送给策略，自动调用策略的on\_bar函数，执行在上一节中讲解的交易逻辑。